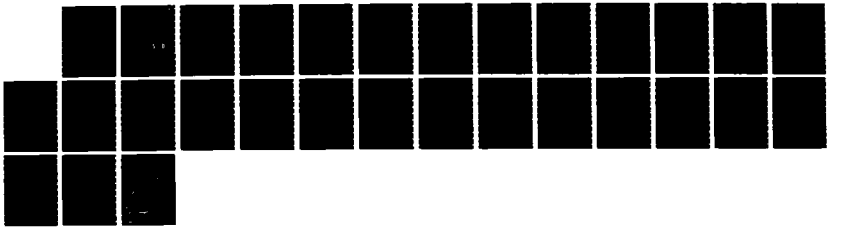
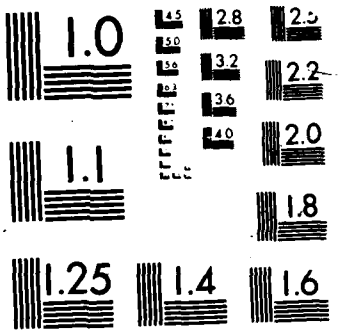
AD-A172 501 LAYER1 A SIMULA CONTEXT FOR SIMULATING THE OPERATION OF 1/1
COMMUNICATION SYSTEMS(U) NAVAL RESEARCH LAB WASHINGTON
DC J P HAUSER ET AL. 05 SEP 86 NRL-8989

UNCLASSIFIED F/G 17/2.1 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

Naval Research Laboratory

Washington, DC 20375-5000 NRL Report 8989 September 5, 1986

# Layer1: A SIMULA Context for Simulating the Operation of Communication Systems

J. P. HAUSER AND D. J. BAKER

*Information Technology Division*



Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE

AD-A172 581

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b RESTRICTIVE MARKINGS |
| 2a SECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited. |
| 2b DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4 PERFORMING ORGANIZATION REPORT NUMBER(S)<br>NRL Report 8989 | 5 MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a NAME OF PERFORMING ORGANIZATION<br>Naval Research Laboratory<br>6b OFFICE SYMBOL (If applicable)<br>Code 7521 | 7a NAME OF MONITORING ORGANIZATION |
| 6c. ADDRESS (City, State, and ZIP Code)<br>Washington, DC 20375-5000 | 7b. ADDRESS (City, State, and ZIP Code) |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION Space and Naval Warfare Systems Command<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State, and ZIP Code)<br>Washington, DC 20363-5100 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. 62721N; PROJECT NO. XF21.222C; TASK NO XF21.242 (STP 3203); WORK UNIT ACCESSION NO. DN080-087 |
| 11 TITLE (Include Security Classification)<br>*Layer1*: A SIMULA Context for Simulating the Operation of Communication Systems | |
| 12. PERSONAL AUTHOR(S)<br>Hauser, J.P. and Baker, D.J. | |
| 13a. TYPE OF REPORT<br>Interim<br>13b. TIME COVERED FROM Jan. 1985 TO Jan. 1986<br>14. DATE OF REPORT (Year, Month, Day)<br>1986 September 5<br>15 PAGE COUNT<br>29 | |
| 16. SUPPLEMENTARY NOTATION | |
| 17. COSATI CODES<br>FIELD / GROUP / SUB-GROUP | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)<br>Communication systems, Object-oriented, Simulation, Carrier-sense multiple-access, SIMULA |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

*Layer1* is a SIMULA class that provides a set of object templates useful in developing simulations of communication system operation. *Layer1* is oriented toward the modeling of high frequency (HF), frequency-hopped, radio communication systems; but it is sufficiently versatile to support the simulation of both radio and hardwired communication systems that use other portions of the frequency spectrum. In this document we describe the *layer1* methodology and give an example of its application to simulating an HF radio communication system that uses a carrier-sense multiple-access (CSMA) protocol. The purpose is to introduce the reader to the object-oriented approach used to develop the *layer1* code and to give him or her enough information to construct a simulation in a *layer1* context.

| | |
|---|---|
| 20 DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS | 21 ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED |
| 22a NAME OF RESPONSIBLE INDIVIDUAL<br>James P. Hauser | 22b TELEPHONE (Include Area Code) (202) 767-2771<br>22c OFFICE SYMBOL Code 7521 |

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

## CONTENTS

# Layer1: A SIMULA Context for Simulating the Operation of Communication Systems

## INTRODUCTION

*Layer1* is a SIMULA context in which communication protocols can be specified and simulated. Its name is derived from the International Standards Organization (ISO) reference model for open systems interconnection (OSI) [1]. *Layer1* of the ISO/OSI architecture specifies the physical means by which information is transported within a communication system. *Layer1* code implements the physical layer of a communication system by providing a set of SIMULA classes (i.e., object templates) that model communication hardware and its interaction with the communication medium. Thus one may define and experiment with communication protocols by using a model of a communication system rather than the actual hardware.

*Layer1* code provides a degree of realism adequate to model frequency-hopped, HF radio communication systems. There are limitations, however. For example, no propagation models are provided to compute communication ranges, although the user can directly incorporate the procedures that perform these computations into the simulation. Also, channels and hop codes can be differentiated, but hopping patterns cannot be defined explicitly. A receiver can detect primary (same channel, same hop code) and secondary (same channel, different hop code) collisions as well as the number of competing signals involved in a collision. However, it cannot determine the actual number of hits (same time, same frequency hop) associated with a collision. Here again, the user can supply procedures to specify hopping patterns and to compute the number of hits if additional detail is required in the model. Thus, the aforementioned limitations can be removed by extending the *layer1* code. The implementation of extensions such as these is straightforward.

In the next section, we briefly describe the supporting context on which *layer1* is built. Then, in the following sections we describe the kinds of objects provided by *layer1*, explain how they work, and show how to use them to design a simulation. In the conclusions we give an assessment of our approach to communication protocol simulation, and in the appendix we list the SIMULA code for a sample problem.

## SUPPORTING CONTEXT

SIMULA [2] provides the foundation for the supporting context in which *layer1* is developed. SIMULA is an extension of ALGOL [3]. It adds to ALGOL a special SIMULA construct called a *class*. SIMULA classes can be used in two distinct ways. One way is to use a class as a context. A context is a precompiled block of SIMULA code containing procedures and object templates that serve as a set of tools and, thus, extend the capability of SIMULA to handle problems in a specific area of interest. Another way to use a class is as an object template. An object template is used as a pattern to create one or more objects of the same type. This is accomplished by using the SIMULA *new* construct, and an object thus created is called a class instance. Object creation (i.e., class instantiation) is illustrated by the following SIMULA statement:

```
xmtr:- NEW transmitter;
```

*Manuscript approved April 8, 1986.

The attribute *xmtr* is a special type of SIMULA variable called a *reference* variable. Reference variables serve as pointers to class instances and must be typed properly somewhere in the SIMULA context. In this case we need a type designation as follows:

```
REF (transmitter) xmtr;
```

which designates *xmtr* as a pointer to *transmitter* objects. The symbol ":-" is read as *denotes* and serves as a replacement operator for reference variables. NEW causes the instantiation to occur. Each class instance in a SIMULA system is an object with its own set of attributes and actions. Therefore, to implement *layer1* we define a set of SIMULA classes that provide templates for the various communication hardware components. The SIMULA code that a user of the *layer1* context writes instantiates the classes as needed to implement his particular communication system model. This is analogous to building a communication system by assembling and interfacing hardware components. We clarify this technique in the sections that follow.

Finally, we rely heavily on the single-inheritance capability of SIMULA classes, in building contexts and in developing object templates. Class inheritance is effected by prefixing one class with another. For example, in Fig. 1 we depict the *layer1* context. The foundation of the context is SIMULA itself. Two standard SIMULA classes, *simset* and *simulation*, are added to SIMULA and together they provide the context available to every SIMULA user. Next, we provide a user-written class called *node_stats*. Class *node_stats* has the following form:

```
simulation CLASS node_stats(max_num_of_nodes);
INTEGER max_num_of_nodes;  !Maximum number of nodes;
BEGIN
  (code which implements class node_stats) ...
END;
```

In the first line of node_stats code we see that the class declaration for node_stats is prefixed by *simulation*. This has the effect of passing on to class *node_stats* the entire context of class *simulation*, which in turn has the entire context of class *simset*, which in turn has the entire context of SIMULA. By the time we define *msg_queue* class *layer1*, *layer1* has inherited the entire context that has come before and in turn may pass it on along with its own classes (i.e., object templates) and procedures to any class that uses *layer1* as a prefix.

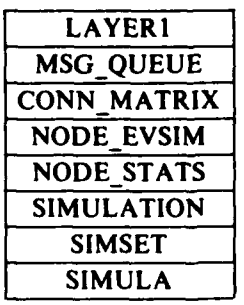



Fig. 1 — Supporting context for *layer1*

Classes that define object templates use inheritance in much the same way as classes that define contexts. For example, let us assume that we would like to extend our model of a transmitter to incorporate features specific to ultrahigh frequency (UHF) transmitters, which are not modeled in the very generalized class *transmitter*. We could do this by defining a new class, *transmitter* class *uhf_transmitter*, which is then called a subclass of class *transmitter*. When class *uhf_transmitter* is instantiated, it will be a concatenated object containing the attributes and actions of both class *transmitter* and class

*uhf_transmitter*. Other varieties of transmitter object templates may also be defined using class *transmitter* as a prefix. Note: the pointer – **REF(transmitter)xmtr** – can point to any concatenated object that uses *transmitter* to begin its prefix chain. Class inheritance provides a powerful and flexible capability for designing object templates as well as contexts.

Figure 1 gives a pictorial representation of the *layer1* supporting context, and a verbal description follows:

- *Simset*, a standard SIMULA class (context), implements queues as two-way linked lists by defining two new classes (templates), *link* and *head*. Any object prefixed by *link* can be inserted into, shuffled around in or removed from any queue that is an instance of class *head*.

- *Simulation*, another standard SIMULA class, supports discrete event simulation. Simulation defines three new classes—*link* class *event_notice*, *link* class *process*, and *process* class *main_program*. Also, an instance of class *head* is created which serves as an event-notice queue. Simulation provides a set of procedures for scheduling event-notices. When an event-notice becomes current, the process referenced becomes active. The main program is also a process with its own event-notice, therefore it becomes another member of the set of quasi-parallel processes that constitutes a SIMULA system.

- *Node_stats* is a class that provides abstract data types for statistics collection [4] and introduces the concept of a node. The introduction of class *node* at this relatively low level in the context facilitates the inclusion of node reference variables and class prefixing in tailoring the statistics collection and other higher level contexts to a node oriented model.

- *Node_evsim* provides an event-process facility [5] that permits a process (i.e., an event-process) to have multiple event notices pending in the event queue. This is an important extension to the SIMULA process as defined in class *simulation*, which permits each process to have only one event notice in the event queue at any one time. Also, *node_evsim* provides event tracing. Event-processes play a significant role in the design of *layer1* code.

- *Conn_matrix* supplies a template and procedures for manipulating connectivity matrices. More about this topic is said later.

- *Msg_queue* extends the basic queue handling facilities of class *simset* to provide procedures *put* and *get* [6] along with an interface to a message processor.

- *Layer1* uses many of the features just described and adds to them its own set of classes and procedures oriented toward the modeling of communication systems.

The entire context described above with all its features and capabilities is available to the *layer1* user.

## LAYER1 OBJECTS

In *layer1* we take advantage of the object-oriented approach afforded by the SIMULA *class* construct. This approach allows us to map directly from real objects, such as transmitters and receivers, to SIMULA classes that represent these objects. By maintaining a one-to-one mapping of real objects to SIMULA objects, we obtain a model that is conceptually identical to the real system we are modeling. Thus, it is easy to understand and use the model. The kinds of object templates (i.e., SIMULA classes) provided by *layer1* are the following:

- **layer1_node**

  A physical platform that can house one or more communication systems.

- **channel**

  A portion of the radio frequency spectrum used to send and receive radio transmissions. *Layer1* channels provide communication paths for chan_msgs to follow.

- **chan_msg**

  The *layer1* transmission unit.

- **iso_msg**

  Contents of a chan_msg.

- **multicoupler**

  The focal point at a layer1_node which receives all transmissions, i.e., chan_msgs.

- **receiver**

  Receives chan_msgs and extracts their information, i.e., iso_msgs.

- **transmitter**

  Packs link layer information, iso_msgs, into chan_msgs and sends them via a channel.

- **controller**

  Manages a suite of transmitters and receivers by running controller_subprograms.

- **controller_subprogram**

  Provides an interface to the transmitter and receiver hardware, which becomes a context for writing link layer protocols.

Figure 2 presents a communication system model constructed with *layer1* objects. These classes are now discussed in detail.

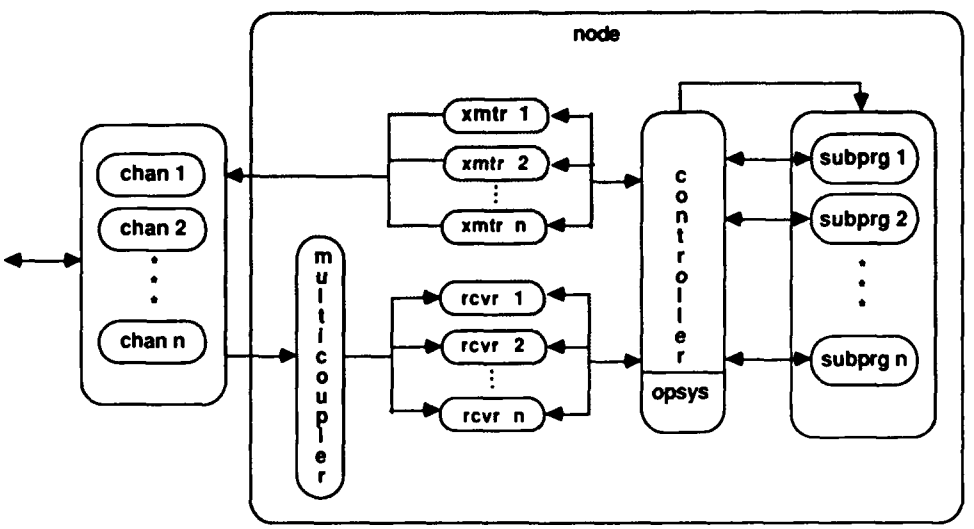



Fig. 2 — The *layer1* model of a communication system

**Layer1_Node**

A layer1_node models a platform that can house one or more communication systems. A layer1_node object is created in the user's program by the following statement:

**nodes** *(index)*:-**NEW** *layer1_node_subclass(id_num,num_xmtrs,num_rcvrs,*
num_cntrls,layer1_node_subclass parameter list);

In this and the statements that follow we use two conventions. The items in **boldfaced** should be typed exactly as shown and the *italicized* items represent quantities, identifiers or code supplied by the user. Class *node_stats* defines REF(node) ARRAY nodes(1:max_num_of_nodes) which is an array of pointers to node objects and is inherited by *layer1*. Since *layer1_node* is a subclass of *node*, nodes pointers may be used to point to layer1_node objects. The user can create nodes(1), nodes(2), etc., up to max_num_of_nodes. That is, *index* assumes values 1, 2, etc., not to exceed max_num_of_nodes. The user specifies max_num_of_nodes when compiling a *layer1* block and thus places an upper bound on the number of nodes that can be simulated with a particular load module. *Layer1_node_subclass* is the name of a subclass of *layer1_node* written by the user; it has the following form:

**layer1_node CLASS** *layer1_node_subclass (parameter list);*
*parameter definitions*
**BEGIN**

*attributes*

*actions*

**END;**

The additional parameter list and definitions as well as the inclusion of additional attributes are optional. We say *additional* because *layer1_node_subclass* inherits all the attributes and actions from its prefix class, which in this example is class *layer1_node*.

We now define the four parameters specified in the class *layer1_node* parameter list. If the user specifies a parameter list for *layer1_node_subclass*, those parameters would be appended to this list. *Id_num* is an integer identification number for the layer1_node being created. It may have the same value as the one used for *index*, but it is not necessary for it to be the same. *Num_xmtrs, num_rcvrs* and *num_cntrls* are integers that specify the number of transmitters, receivers, and controllers to be created at this node. In each layer1_node_subclass object, the user must create the transmitter, receiver, and controller objects residing at that layer1_node and must tell *layer1* how to interconnect them. These actions are appended to the actions of class *layer1_node*, which create a multicoupler object. The details of these actions are explained as we discuss other classes.

We close our discussion of layer1_nodes with one last point; layer1_nodes need not be all alike. Of course, differing layer1_nodes will require differing layer1_node subclasses (e.g., *layer1_node_subclass_1, layer1_node_subclass_2*, etc.). The user creates as many of each kind of layer1_node objects as required. Creation of different layer1_node objects and assignment of pointers to array *nodes* is accomplished in just the same way as shown above.

**Channel**

Layer1_nodes are connected to each other via communication channels. *Layer1* provides a class *channel*. Channel objects can be created with the following procedure call:

**create_phys_chans** *(num_of_chans,dynamic);*

*Num_of_chans* is an integer that specifies the number of channels to be created. When a channel is created, a pointer is passed to REF(channel) ARRAY chans(1:max_num_of_chans). The upper array

bound, max_num_of_chans, is set by the user when compiling a *layer1* block and limits the number of channels that may be created with that particular load module. *Dynamic* is a Boolean number that determines whether or not the channels will be dynamic. A dynamic channel may alter its connectivity matrix at any time during the course of a simulation. Otherwise the channel is static and altering its connectivity matrix will produce erroneous results. The only advantage of using static channels is to save computation time. Figure 3 depicts a connectivity matrix.

| receiving \ transmitting | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 |

Fig. 3 — Example of a connectivity matrix for a four node network

The row position tells which node is receiving, and the column position tells which node is transmitting. A "1" indicates connectivity, a "0" indicates lack of connectivity. For example, the 1 at position-1,3 indicates that node-1 can hear node-3. Note that the matrix diagonal contains 0's. This means the nodes for this example are not self jamming. Node pairs that have full two-way connectivity are (1,2), (1,3), and (2,4). One-way links exist from 3 to 4 and from 4 to 1. Each channel creates a connectivity matrix of its own, but it does not initialize it. The user can call an initialization procedure with the following statement:

**chans** *(index)* **.conn.fill_connectivity_matrix;**

Execution of this statement will cause the channel specified by the integer *index* to be initialized by prompting for input data via the user's terminal. It is possible to use an alternative technique for matrix initialization. For example, one might wish to compute the connectivity matrix via a propagation model rather than enter the contents of the connectivity matrix manually. Since the procedure fill_connectivity_matrix is a SIMULA virtual procedure, it can be virtually redefined in a subclass of *intmatrix* class *connectivity* to supply the alternate technique. The SIMULA virtual declaration is illustrated by the following line of code:

**VIRTUAL: PROCEDURE fill_connectivity_matrix;**

This virtual declaration is the first declaration in the body of class *connectivity*, which is a subclass of class *intmatrix*. *Connectivity* is therefore said to be *inner* to class *intmatrix*. Because fill_connectivity_matrix has been virtually declared in class *connectivity*, it becomes an attribute of that class and is accessible via dot notation just as any other attribute of class *connectivity* would be. However, because of the virtual declaration, the attributes and actions of procedure fill_connectivity_matrix need not be specified in class *connectivity*. Rather, they may be specified in a subclass of *connectivity*. SIMULA will use the innermost specification for a virtual procedure. This makes it possible to define a default specification for fill_connectivity_matrix in class *connectivity*, and then redefine it in a class inner to *connectivity*. In fact, a virtually redefined procedure may itself be virtually redefined in a procedure inner to it since SIMULA uses the innermost specification.

The procedure for computing propagation delays is also a SIMULA virtual procedure. The default procedure defined as a *layer1* global procedure yields a value of 0.0 for the propagation delay. The default definition follows:

**REAL PROCEDURE prop_delay (n1,n2); REF(node)n1,n2; prop_delay:=0.0;**

If one wishes to use nonzero propagation delays, this procedure may be redefined in a subclass of *layer1*. The arguments passed to prop_delay are pointers to the transmitting node and the receiving node.

Class *channel* declares two virtual procedures. Their default definitions follow:

**PROCEDURE chan_stat(t); VALUE t; TEXT t;;**

**PROCEDURE chan_graf(t); VALUE t; TEXT t;;**

If redefined, these procedures offer a convenient way to collect statistics (chan_stat) or implement graphics (chan_graf) without making modifications directly to the *layer1* code. Every time a channel processes an event, these procedures are called. The code written in these procedures can respond to every event to which a channel object responds in order to update statistics variables or execute graphics commands. In *layer1*, almost all state changes occur as a result of events being generated and sent to event-process objects [5]. Reference 5 explains event-processes in detail and lists the event-process code. (Other event-process classes in *layer1* besides *channel* are *transmitter*, *receiver*, *multicoupler*, and *controller_subprogram*). Thus the code the user writes for these procedures is not handicapped by being written externally to *layer1*. Separate procedures for statistics collection and graphics are called not out of necessity but rather as a means of modularizing the code. Transmitter and receiver objects call equivalent procedures for the same purpose.

**Chan_Msg**

Class *chan_msg* provides a template for the *layer1* communication unit. Chan_msgs are never used directly. Instead, *layer1* transmitter objects create chan_msgs and send them. When a chan_msg is created, the following chan_msg attributes are set:

- **channel**

  An integer that designates which channel the transmitter sending the chan_msg is tuned to.

- **fhcode**

  An integer that designates which frequency-hop code the transmitter sending the chan_msg is set to.

- **xmtr**

  A ref(transmitter) pointer to the transmitter sending the chan_msg.

- **numofinfobits**

  An integer that specifies the length of the chan_msg. The length of a chan_msg is determined from the length of the iso_msg contained in the chan_msg. This will be explained shortly.

- **data**

  A ref(iso_msg) pointer that points to the iso_msg contained in this chan_msg.

- **msg_id**

  Every chan_msg is assigned a unique integer message *id* to facilitate tracing.

- **origtime**

  Every chan_msg is also marked with the simulation clock time (real) when created.

**Iso_Msg**

Since chan_msgs exist only within *layer1*, we need another type of message object to pass information through the interface to *layer1*. This is the purpose of class *iso_msg*. The only attribute of class *iso_msg* is an integer called *mlength*; it specifies the length of the iso_msg. *Iso_msg* must be used as a prefix for any class of message objects using the *layer1* interface. The SIMULA code presented in the appendix illustrates this technique.

**Multicoupler**

The multicoupler receives all chan_msgs sent to a node and routes them to each receiver that is tuned to hear them. This is determined by comparing the channel to which each receiver at the layer1_node is set with the channel of the incoming chan_msg. The multicoupler also updates variable arrays used for collision detection. The values in these arrays may be read by using the following procedures:

- **num_prim_collisions** *(chan,code);*

  The attribute *chan* is the index number of the channel, and the *code* is the index of the frequency-hopped code. Num_prim_collisions returns the number (integer) of competing signals in the channel and on the same code.

- **num_scnd_collisions** *(chan);*

  The attribute *chan* is the channel index number. Num_scnd_collisions returns the number (integer) of competing signals in the channel including those on different codes.

Each layer1_node creates its own multicoupler without assistance from the user.

**Receiver**

To create and properly initialize a receiver object requires the following code:

**rcvr** *(num)* :- **NEW receiver(** *receiver object title* **, THIS layer1_node** , *num* **);**

*Num* is an integer in the range $1 \leqslant num \leqslant$ num_rcvrs where num_rcvrs is the number of receivers specified in the argument list for the layer1_node. Receiver objects should be created by the layer1_node to which they belong. The statement given above should be one of the actions of a subclass of *layer1_node* as explained in section "Layer1_Node". The *receiver object title* is a string (SIMULA text object) that will be used for identification if event tracing is requested. We recommend a title similar to the following:

**merge_text ("receiver** *num* **at node " , int_text(id_num))**

We use procedure merge_text to concatenate the text items into one object and procedure int_text to convert from integer to text.

Once a receiver is created, another action is also necessary that tells the receiver to which controller it is connected. For that purpose, the receiver has a procedure that must be called as follows:

**rcvr** *(num)* **.identify_cntrl** *(cntrl_num)* ;

A receiver may be connected to only one controller, although a controller may have more than one receiver. *Cntrl_num* is an integer in the range $1 \leqslant cntrl\_num \leqslant$ num_cntrls (section "Layer1_Node").

After having created a receiver and having assigned it to a controller, it is still necessary to *activate* it. Activation of an event-process, such as a receiver object, performs the object initialization tasks and prepares it to receive events. Transmitters, controllers, and controller_subprograms also must be explicitly activated, as it will be discussed later. The following statement will activate a receiver object:

**ACTIVATE rcvr** *(num)* ;

The user interacts with receiver objects via an interface provided by a controller_subprogram object (section "Controller"). This gives the user access to the following procedures for controlling receiver objects:

- **select_rcvr** *(id);*

  The argument *id* is an integer that specifies the receiver (as given by *num* above) one wishes to access by means of the interface. All procedure calls following the call to select_rcvr deal specifically with the receiver named by *id* until select_rcvr is called with a new *id*. If there is only one receiver, the select_rcvr procedure need not be called.

- **read_rcvr_num;**

  This integer procedure returns the index *(id)* of the receiver that is currently selected (i.e., for which the interface is currently active).

- **set_rcvr_channel** *(c);*

  The argument *c* is an integer that selects the new channel to which the receiver is tuned. The value given to *c* corresponds to the channel index as described in section "Channel".

- **rcvr_channel_num;**

  This integer procedure reads the channel to which the receiver is presently tuned and returns the channel index value.

- **set_rcvr_fhcode** *(f);*

  The argument *f* is an integer that selects the receiver's hop code. The value of *f* is compared with the hop code value in chan_msg as set by the transmitter sending the chan_msg. If the values are the same, then the receiver can receive the chan_msg. One does not have to use hop codes. If this procedure and the corresponding procedure for the transmitter (set_xmtr_fhcode) are not called, all hop code values default to 1. Thus, the comparison test mentioned above will always be true, in effect it eliminates any dependency on hop codes.

- **rcvr_fhcode;**

  This integer procedure returns the current value of the receiver's hop code.

- **rcvr_in_sync;**

  This Boolean procedure returns a value of true if the receiver is in sync and false if it isn't.

- **collision_detected;**

  This Boolean procedure returns a value of true if a collision state has occured since the last time the collision flag was cleared.

- **clear_collision_flag;**

  A call to this procedure clears the collision flag.

- **set_scnd_collision_lim** *(l);*

  The argument *l* is an integer that gives the number of secondary collisions (same channel but different codes) that a receiver can tolerate.

- **rcvr_collim;**

  This integer procedure returns the current collision limit setting.

- **num_prim_signals;**

  This procedure call returns the number of primary signals currently being received.

- **num_scnd_signals;**

  This procedure call returns the number of secondary signals currently being received.

The procedures listed above form a subset of commands that may be used to program a communication controller. The commands just given control receivers. We introduce the commands for transmitters in section "Transmitter" and discuss more general concepts and additional commands in section "Controller."

Class receiver provides several virtual procedures that may be redefined. Rcvr_stat and rcvr_graf are the counterparts of chan_stat and chan_graf that were discussed in detail in section "Channel." In addition, class receiver provides two Boolean virtual procedures that may be redefined. Their default definitions follow:

```
BOOLEAN PROCEDURE collision_test;
INSPECT station.mltcplr DO
collision_test:=(IF num_prim_collisions(rchannel,rfhcode)>1 THEN TRUE
ELSE num_scnd_collisions(rchannel)>scnd_collision_lim);

BOOLEAN PROCEDURE cannot_sync;
cannot_sync:=FALSE;
```

The default procedure for collision test inspects the station's (i.e., layer1_node's) multicoupler in order to access the counters that contain current state information on the number of primary and the number

of secondary collisions. If a primary collision state exits (num_prim_collisions>1) or if the number of secondary collisions exceeds the secondary collision limit, synchronization with any transmitter can neither be achieved nor maintained. If this does not adequately model the performance of the receiver in a collision state, procedure collision_test may be redefined in a subclass of receiver. There may exist other conditions in a real receiver besides the collision state that could preclude synchronization. For example, it might be necessary to set the receiver for the proper transmission rate before it can achieve synchronization. *Layer1* receiver objects can be tailored to respond to other sets of synchronization conditions by virtually redefining procedure cannot_sync. The default procedure shown above always returns a false value; therefore, it will never interfere with a receiver's ability to synchronize.

The last virtual procedure contained in class *receiver* is real procedure time_to_sync. The default definition is as follows:

**REAL PROCEDURE time_to_sync; time_to_sync:=0.0;**

The value returned by a call to time_to_sync is used to schedule an event that synchronizes the receiver. Section "Layer1 Events" explains this more fully. To obtain a nonzero synchronization time, procedure time_to_sync must be virtually redefined in a subclass of receiver.

### Transmitter

Transmitter objects are created, initialized, and activated in the same fashion as receiver objects were. The appropriate code is:

**xmtr** *(num)* **:- NEW transmitter(** *transmitter object title* **, THIS node ,** *num* **);**

**xmtr** *(num)* **.identify_cntrl** *(cntrl_num)* **;**

**ACTIVATE xmtr** *(num)* **;**

Just as with receiver objects, the user accesses transmitter objects by means of a controller_subprogram interface that provides the following procedures:

- **select_xmtr** *(id);*

  The argument *id* is an integer that designates which transmitter the interface is currently active for. If there is only one transmitter, this procedure need not be called.

- **read_xmtr_num;**

  This integer procedure returns the current value of *id.*

- **set_xmtr_channel** *(c);*

  The argument *c* is an integer that designates the channel being selected. *C* is used as an index to REF(channel) ARRAY chans(1:max_num_of_chans) which is an array of pointers to channel objects.

- **xmtr_channel_num;**

  This integer procedure returns the index of the channel currently used by the transmitter.

- **set_xmtr_fhcode** *(f);*

  The argument *f* is an integer that designates the frequency hop code being selected.

- **xmtr_fhcode;**

  This integer procedure returns the designator of the frequency hop code currently used.

- **set_xmtr_info_rate** *(r);*

  The argument *r* is an integer that designates the new transmission rate selected. Information bits/s would be an appropriate choice of units in many cases; however, another choice of units would be acceptable. The choice should be compatible with the unit of length chosen for iso_msgs.

- **xmtr_info_rate;**

  This integer procedure returns the current transmission rate.

- **turn_on_xmtr;**

  This procedure turns the transmitter on. It does not initiate the transmission of information, but it does send a carrier that initiates receiver synchronization and can be collision detected by receivers that have connectivity with the transmitter.

- **turn_off_xmtr;**

  This procedure turns the transmitter off.

- **start** *(msg);*

  The argument *msg* is an iso_msg that is packed in a chan_msg and begins transmission at the moment this procedure is called. The transmitter must be in an idle state (i.e., turned on and not sending another chan_msg) for this procedure call to take effect. If a start is attempted when the transmitter is not idle, a warning message is printed.

Class *transmitter* has two additional procedures that may be virtually redefined—xmtr_stat and xmtr_graf. These procedures are analogous to chan_stat and chan_graf previously discussed in the section "Channel".

### Controller

A controller object manages a set of communication assets; receivers and transmitters, and grants controller_subprograms access to these assets according to a user specified protocol. Process class controller contains two interfaces. The first is an interface for controller_subprograms to use in accessing transmitter and receiver objects. A controller_subprogram uses this interface to create the interface presented to the user as discussed in the sections: "Multicoupler," "Receiver," and "Transmitter." The other interface is designed to be used by an operating system that has the task of scheduling the controller_subprograms assigned to the controller. The operating system is written as a subclass of controller. The simplest example is that of an operating system which initiates the execution of one subprogram and after that does nothing more as shown here:

```
controller CLASS opsystem;
BEGIN
 subprg(1):-NEW subprog(merge_text("subprog for cntrl[1] at node ",
        int_text(station.id_num)),station,"subprog",THIS controller);
 ACTIVATE subprog;
 run_sp("subprog");
END of opsystem;
```

The code to create and activate the controller object just discussed is the following:

```
cntrl (num) :- NEW opsystem (THIS node , num );

ACTIVATE cntrl (num);
```

The new object thus created is a subclass of controller. The actions of controller, which link the controller with its transmitters and receivers and initialize the interface to be active for transmitter *1* and receiver *1*, are executed first, and then they are followed by the actions of opsystem.

In a more complicated example, an operating system program that regularly swaps two controller_subprograms (say, subprog1 and subprog2) with period *2t* can be written as follows:

```
controller CLASS opsystem(t); REAL t;   ! t is time to run before swapping;
BEGIN
 subprg(1):-NEW subprog1(merge_text("subprog1 for cntrl1 at node",
        int_text(station.id_num)),station,"subprog1",THIS controller);
 subprg(2):-NEW subprog2(merge_text("subprog2 for cntrl1 at node",
          int_text(station.id_num)),station,"subprog2",THIS controller);
 ACTIVATE subprog(1);   ACTIVATE subprog(2);

 loop:
 run_sp("subprog1");
 HOLD(t);
 run_sp("subprog2");
 HOLD(t);
 GOTO loop;
END of opsystem;
```

The first two actions of opsystem are to create subprog1 and subprog2 and pass their pointers to REF(controller_subprogram) ARRAY subprg(1:max_num_of_subprograms). Procedure run_sp uses these pointers to halt the currently running subprogram and start running the subprogram named as run_sp's argument. *Hold* is a class *simulation* procedure that schedules opsystem for reactivation at a simulation time equal to current simulation time plus time *t*, which is hold's argument. Thus, execution of *loop* causes subprog1 to be activated and then, after time *t*, subprog1 to be halted and subprog2 to be activated. After another time *t* the cycle repeats itself ad infinitum until the simulation is terminated.

Besides run_sp the operating system interface also provides a procedure called halt_sp. Run_sp actually calls halt_sp to halt the currently operating subprogram before activating the next one. However, halt_sp may be called independently, in which case no subprogram will be running in the controller.

**Controller Subprogam**

*Layer1* can model the case of several networks sharing the same set of transmitters and receivers. For example, as in the case of HF Long Haul and HF Intrabattle Group Networks sharing the same HF

communications suite, the controller acts as the arbiter that says which network has access to the communication assets at the current time. Each network has a controller_subprogram associated with it that implements the link layer protocol for the corresponding network. Protocols for higher layers may also be built on top of the link layer protocol if desired.

In sections "Multicoupler" and "Receiver" we have discussed many of the procedures that form the controller_subprogram interface. However, the preceding discussions are not complete without mentioning three procedures that **must** be virtually redefined in a subclass of controller_subprogram – msg_sent, msg_ret and msg_rcvd. These three procedures are called by *layer1* and their actions are specified by the user in a subclass of controller_subprogram. A description follows:

- **msg_sent(iso_msg,tid);**

  When a transmitter completes its transmission of an iso_msg (as the contents of a chan_msg), the transmitter calls this procedure with a pointer to the iso_msg just sent and the transmitter's integer index as its arguments. This informs the user's protocol (as implemented in the controller_subprogram subclass), that the iso_msg it previously started has been sent and now it is time to initiate another action—perhaps to get another iso_msg out of a queue and start transmitting it or perhaps to turn off the transmitter.

- **msg_ret(iso_msg,tid);**

  In the event a transmitter is turned off while an iso_msg is being transmitted, the transmitter will call this procedure. Perhaps the user would like to place the aborted iso_msg in a retransmission queue. If so, the msg_ret procedure can be programmed for that action. Perhaps the user would like to discard the iso_msg without taking any further action. In that case, msg_ret may be defined without any actions.

- **msg_rcvd(iso_msg,rid);**

  This procedure is called by a receiver when a chan_msg is received. The iso_msg is unpacked from the chan_msg to become the actual parameter for the call to msg_rcvd. Chan_msgs are demarcated by start_of_msg and end_of_msg events. Msg_rcvd is not called until the end_of_msg event is received. A call to this procedure informs the user's controller_subprogram subclass that the receiver with index number *rid* has received an iso_msg.

The three procedures described above complete the set of procedures provided by controller_subprogram as an interface to the *layer1* transmitter and receiver objects.

Controller_subprogram defines another procedure that may be called by the user, but it is not a part of the interface to transmitter and receiver objects. The procedure is named *designate_clock* and is called as follows:

**subprg** *(num)* **.designate_clock** *(process name)* ;

The process object pointed to by *process name* serves as a clock mechanism. It is not necessary to associate a clock with a controller_subprogram, but it can be useful. Since it is a process class, the clock may use procedure hold to schedule synchronous events for its controller_subprogram. Of course, controller_subprogram is also a subclass of process, but, since it is an event-process, a call to hold will yield unpredictable results [5]. Therefore, to schedule synchronous events for a controller_subprogram, one should create a separate process object to serve as a clock. However, if the controller is running more than one controller_subprogram, the clock should be running only while its associated controller_subprogram is executing. Stopping and starting a clock mechanism is handled behind the

scene by procedures in controller_subprogram as long as the procedures have access to a pointer that designates the clock—hence, procedure designate_clock.

This concludes our discussion of *layer1* objects. In a following section we study an example of a carrier-sense multiple-access protocol written in the *layer1* context, which illustrates most of the material presented in this section. However, before launching into the example, we wish to explain how *layer1* uses events to model the transmissions of a communication system. This we endeavor to do in the next section.

## LAYER1 EVENTS

*Layer1* uses events to model the transmission process in a communication system. By understanding how the occurrence of one event leads to the occurrence of other events and how it produces state changes in the communication system model, one is able to see how *layer1* works. To aid in gaining this understanding, we now introduce the concept of an event graph [7]. We take some liberty with event graphs as they are formally presented in Ref. 7, and therefore give the following explanation of the conventions we use in this report.

Figure 4 shows our event graph conventions. Events are depicted as circles connected by directed edges. Each circle is tagged with the event name and, in some cases, the lower half of the circle is used to indicate a new system state. All events create new system states; however, we do not always rigorously spell out these new states in the event diagrams. Where a new state is indicated, it is entered after the occurrence of the event, not before.

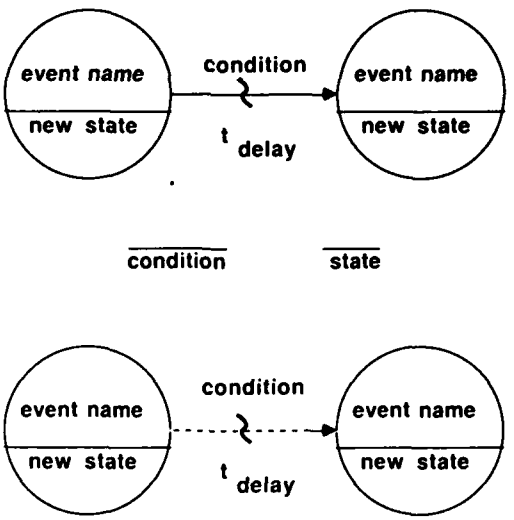



Fig. 4 — Event graph conventions

Directed edges indicate causality between events. Edges may be tagged with conditions and/or delay times. If the edge is tagged with a condition it is marked with a "~" to indicate its conditional nature. Conditions always represent some combination of state variables (i.e., system state) which must be satisfied for an event to occur. Moreover, the condition must be satisfied at the time of occurrence of the causing event, not after it has occurred. Thus, the new state indicated in the lower half of the causing event's circle has not yet been entered at the time the condition must be tested. If a delay time is given, the event thus scheduled will occur at a time equal to the time of occurrence of the causing event plus the delay time.

Two more conventions complete our set of event graph tools. If a state or a condition has a line above it, the opposite is indicated and may be read as "not." Also, a dashed edge indicates the canceling of an event as opposed to the scheduling of an event.

With these event graph tools we are now ready to diagram the workings of *layer1* objects, that are presented in Figs. 5 and 6. The process of transmitting a message is shown in Fig. 5, and the process of receiving a message is shown in Fig. 6.

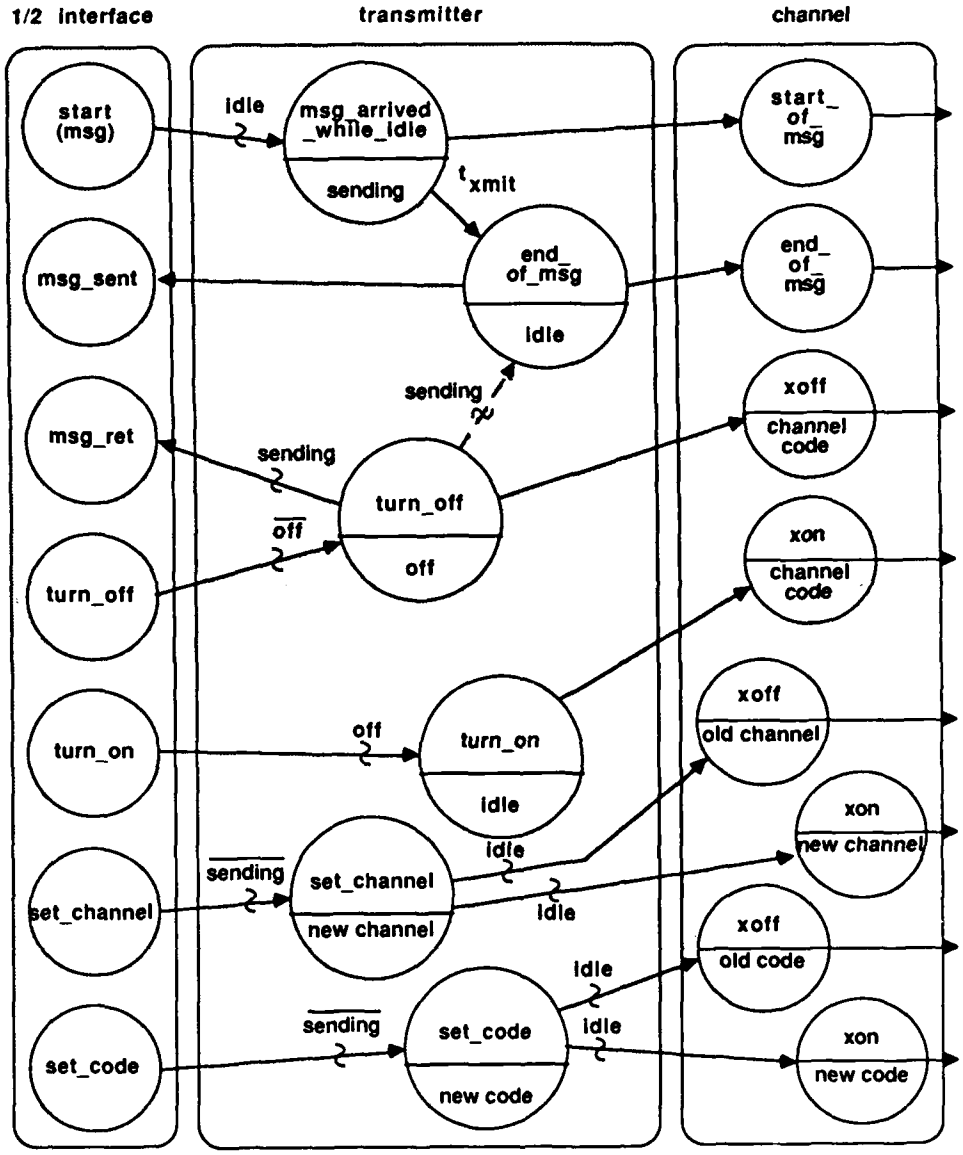



Fig. 5 — Event graph of *layer1* transmitter and channel

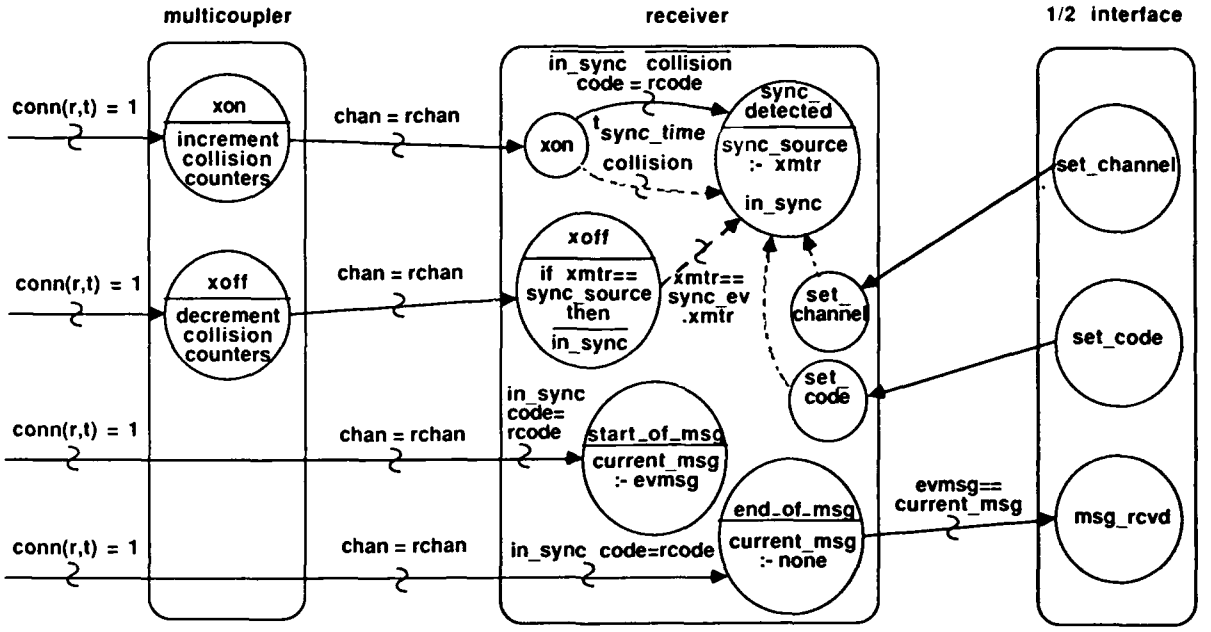



Fig. 6 — Event graph of *layer1* multicoupler and receiver

In Figs. 5 and 6 we use one other convention not directly associated with event graphs, that is the use of boxes to denote objects. The box at the far left of Fig. 5 represents a controller_subprogram object. The events or procedure calls* listed there are scheduled by a user defined protocol that must implement some kind of channel access scheme. This box is the interface to *layer1*. We have not shown the interface in its entirety here in order to avoid unnecessary clutter in the event diagram. The parts of the interface essential for understanding *layer1* message transmission are included.

The real work of transmitting an iso_msg is done by the box in the middle, the transmitter object. If we ignore channels and codes for a moment, we see that a transmitter object has three fundemental states — sending, idle, or off. All edge conditions are related to one of these states. For example, to start a message the transmitter must be in the idle state. The idle state means that the transmitter has been turned on but is not sending any information. One could view it as a transmitter sending an unmodulated carrier. If the transmitter has not been turned on or if it is in the process of sending a message, then calling start with a new iso_msg to send will have no effect, except that *layer1* will return a warning message to the user appraising him of the situation. But, if the transmitter is idle, starting a message immediately queues a msg_arrived_while_idle event which, in turn, schedules unconditionally a start_of_msg event in the channel and an end_of_msg event delayed by the message transmission time in the transmitter. The transmitter state is then set to sending. Upon completion of the transmission the transmitter end_of_msg event occurs and in turn schedules an end_of_msg event for the channel and calls the virtual msg_sent procedure back in the controller_subprogram interface where the user's protocol must decide what to do when a message has been sent. The actions to be taken upon completion of a message transmission become the actions programmed in the virtual definition of the msg_sent procedure.

* The uniqueness of an event is that it may be scheduled for activation in a time-ordered event-queue. When an event gets to the top of the event-queue, and thus becomes current, the actions it causes become the active part of the simulation. This is precisely what happens when a procedure call is made — i.e., the actions of the procedure become the active part of the simulation. Therefore, in a special case when an event is scheduled for immediate activation, it behaves not differently than a procedure call. In fact, a procedure call can accomplish the same result as scheduling an event for immediate activation in the event-queue. The controller_subprogram interface happens to be implemented with procedure calls, but it is not improper to graph these procedure calls as events for the reason just given.

Calls to turn_off and turn_on in the controller_subprogram interface conditionally schedule events by the same name in the transmitter. The rationale for imposing these conditions should be fairly obvious. If the transmitter is already off, a call to turn_off should have no effect. Thus, the transmitter has to be in a "not" off state (i.e., either idle or sending) to schedule a turn_off event. Conversely, the transmitter must be off for a turn_on event to be scheduled. Both turn_off and turn_on events unconditionally schedule corresponding events in the channel. However, if a transmitter is turned off while it is sending a message, a little extra work must be done to tidy things up. The end_of_msg event which had been scheduled for the end of the transmission must be canceled and a virtual procedure msg_ret is called to force the user's protocol to do something about the aborted transmission. Of course, if the user merely wants to allow the aborted message to fall on the floor, his virtual redefinition of msg_ret need take no action.

The set_channel and set_code events may only be scheduled if the transmitter is not sending, i.e., is off or idle. When code or channel changes are made while the transmitter is in an idle state, additional xoff and xon events must be scheduled in the channel so that receiving nodes can sort out what is going on. Xon and xoff events are always tagged with the channel and code of the transmitter scheduling them.

The channel object has the task of forwarding all start_of_msg, end_of_msg, xon, and xoff events to the multicouplers at those nodes with which the transmitter using the channel has connectivity. If the channel happens to be designated as *dynamic* the task is slightly more complicated because the channel must generate some xon's and xoff's of its own to model the effects of changing connectivities. However, this is done behind the scene so that the user need not be concerned with it. The user needs only be concerned with keeping the connectivity matrix up to date.

In Fig. 6 we diagram the effects of the start_of_msg, end_of_msg, xon, and xoff events as they are received. Reception is contingent upon having connectivity with the sending transmitter by means of a channel, as the conditions used to tag the far left edges in Fig. 6 indicate. The multicoupler intercepts xon and xoff events to increment and decrement collision counters within the multicoupler. Events are then forwarded to any receiver at the node tuned to the channel that is sending the events. If the receiver is not already in sync and is not in a collision state, the xon event will schedule a sync_detected event delayed by the time required to obtain synchronization. The length of time required is computed by a call to the procedure time_to_sync, which may be virtually redefined in a subclass of receiver (section, "Receiver"). Xon and xoff events can conditionally cancel the sync_detected event at a receiver. If the xon event causes a collision state to occur at the receiver, then the receiver is not able to attain synchronization. Thus the sync_detected event is canceled. Also, if the transmitter that caused the sync_detected event to be scheduled sends an xoff, the sync detected event must be canceled. Changing the channel and code settings of a receiver also cancels a sync_detected event that has been scheduled. Start_of_msg and end_of_msg events can be scheduled at the receiver only if the receiver is set to the proper code as well as the proper channel and is in sync, as is indicated by the edge conditions. The start_of_msg event sets a reference variable called current_msg to point to the incoming chan_msg. The end_of_msg event calls a virtual procedure msg_rcvd in the controller_subprogram interface as long as the chan_msg causing the end_of_msg event is the same one that caused the start_of_msg event. The user's protocol then must decide what to do with a message that has been received by redefining the virtual procedure.

## AN EXAMPLE

To illustrate the use of the *layer1* context we show how to code a simple carrier-sense multiple-access (CSMA) protocol. An event diagram of the protocol is shown in Fig. 7. To develop a *layer2* protocol we do not need to know anything about *layer1* other than how to use its interface. In the adjacent box we show the procedures we shall use from the interface to implement the protocol. Over on the right-hand side of the diagram we have another interface—the 2-3 interface. The 2-3 interface contains the procedures we want our network layer to use. We model the network layer and above very

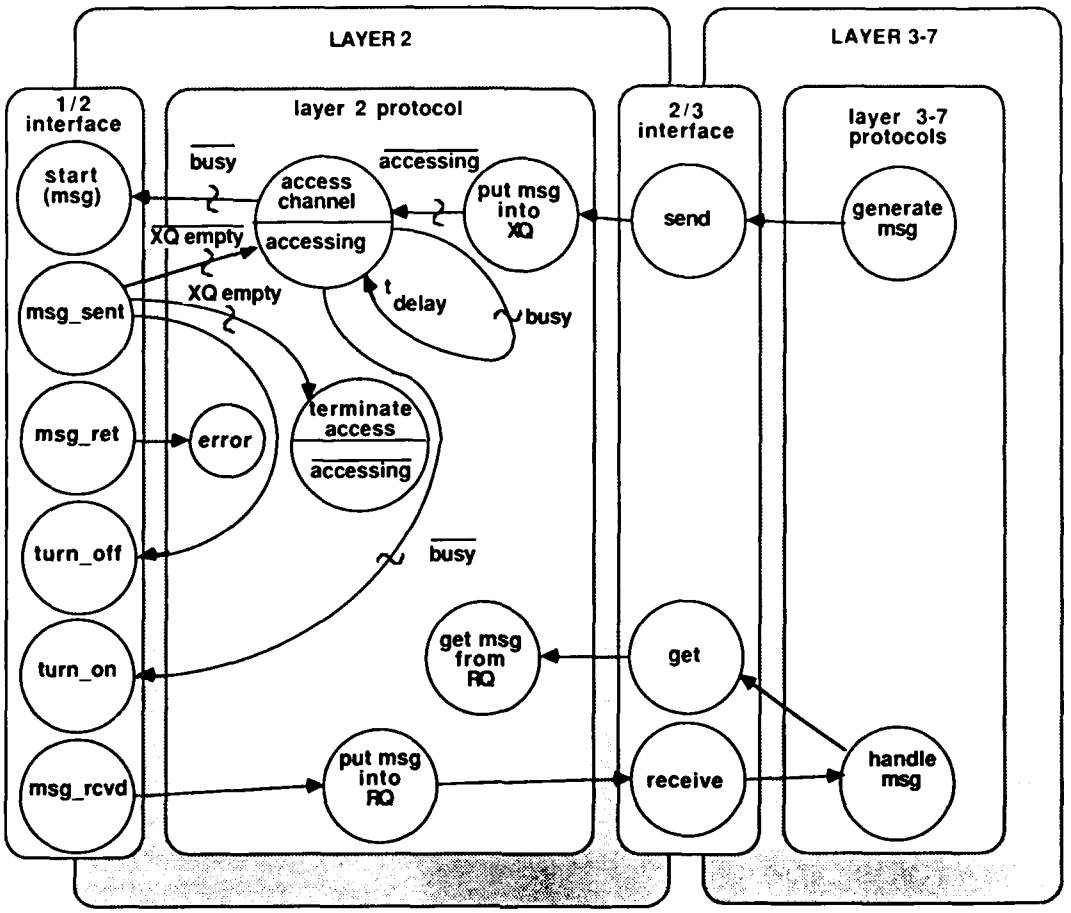



Fig. 7 — Event graph of a CSMA protocol

simply by creating a traffic generator to generate the message traffic at each node and by creating a message handler that merely gets incoming messages and prints them. To design the protocol, therefore one should simply fill in the box between the two interfaces with a meaningful event diagram, as we have done.

Now we are ready to examine the CSMA protocol. If we wish to send a message via the 2-3 interface, we call the 2-3 interface send procedure. A call to this procedure puts the message in a queue — xq — and schedules an access_channel event if the protocol is not already in an accessing state. When an access_channel event occurs, the protocol checks to see if the channel is idle (not busy). If it is, the protocol turns on the transmitter and starts a message — the first message in xq, which is an FIFO queue. If, however, the channel is busy, the access_channel event reschedules itself to occur at some random delay time later. In either case, the protocol enters an accessing state, which means that it is attempting — either successfully or unsuccessfully — to access the channel.

The rest of our CSMA protocol is implemented by redefining the virtual procedures msg_sent, msg_ret, and msg_rcvd. When the transmission of a message is completed, we must decide what to do next by appropriately redefining the msg_sent procedure. At this point we turn off the transmitter and, if xq has more messages, we schedule an access_channel event. If xq is now empty however, we schedule a terminate_access event which changes the protocol's state to not accessing. In other words, once the protocol gains access to the channel, it keeps on sending messages until xq is empty. This protocol will tend to maintain a high throughput with some sacrifice of message delay time. The protocol could likely be enhanced, but we only intend for it to illustrate simulation design techniques not optimum protocol design.

The definitions of the remaining virtual procedures in the 1-2 interface are quite simple. Msg_ret merely prints out an error message. If our code properly implements the event diagram we should never see this message. If we do see it, it will aid in debugging the code. Msg_rcvd puts the incoming message into an FIFO queue – rq – and queues an event for the next higher protocol that causes it to respond to the incoming message. The code that implements this event diagram is presented in the appendix.

## CONCLUSIONS

The work presented in this report has several significant aspects. The first is the development of the *layer1* code and the context that supports it. *Layer1* has the capability of modeling the physical layers of many different types of communication systems. Moreover, this capability is accessible through an easy to use interface. Secondly, and perhaps of greater significance, is that the techniques used to construct the *layer1* code can be readily extended to simulate the protocols of each of the higher layers of the ISO/OSI reference model. Each new class thus written can become the supporting context for the next higher ISO layer protocol. This creates an exact parallel between the code that implements the simulation and the system being modeled. Finally, by using event graphs we have introduced a very useful technique for describing the simulation model apart from writing the code. A careful inspection of the code presented in the appendix shows that the event graph of our CSMA protocol maps rather directly into SIMULA code. We believe the methodology presented in this report to be both unique and powerful.

## REFERENCES

1. "Reference Model for Open Systems Interconnection," *ISO/TC97/SC16* (Doc. N227), June 1979.

2. G.M. Britwistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*, Studentlitteratur, (Lund, 1977).

3. B. Randell and L.J. Russell, *ALGOL 60 Implementation* (Academic Press, 1964).

4. C.E. Landwehr, "An Abstract Type for Statistics Collection in SIMULA," *ACM Trans. Programming Languages and Systems* **2**, 544-563 (1980).

5. D.J. Baker and J.P. Hauser, "An Event-Process Facility Built on SIMULA: A Tool for Simplifying the Simulation of Distributed Control Systems," *SCSC' 85*, Chicago, July 22-24, 1985. Paper presented at conference and published in the proceedings.

6. K. Babcicky, *Activate SIMULA*, NCC Publication S.88 (1977).

7. L. Schruben, "Simulation Modeling with Event Graphs," *Commun. ACM* **26**, 957-963 (1983).

## APPENDIX

The code for our example is written as two separate blocks. The first is *layer1* class *layer2*. Thus, *layer2* is developed in a *layer1* context. Most of the code defines new subclasses of *layer1* object templates. Also, we redefine virtual procedure prop_delay.

- **real procedure prop_delay**

  To model the effects of propagation delay we must redefine this procedure. The code used in our example reads in a single propagation delay value to be used for all node pairs.

- **iso_msg class csma_msg**

  Here we define a subclass of iso_msg. We merely wish to add a few attributes to iso_msg – contents, ser_no and origtime. The contents of a csma_msg is a net_msg. Net_msgs originate in the layer above *layer2* and become the contents of *layer2* messages, i.e., csma_msgs. In the same way csma_msgs become the contents of chan_msgs, which are the messages of *layer1*. Csma_msgs also have their own serial numbers and origination times.

- **event_msg class net_msg**

  Net_msg must be defined in *layer2* so that *contents*, which is a formal parameter of class *csma_msg*, may be properly typed as a net_msg reference variable. Net_msg is prefixed with event_msg enabling us to use net_msgs as arguments for events. We have not used that capability in this example. However, prefixing message classes with *event_msg* can make the code easier to extend.

- **layer1_node class link_node**

  Link_node has the task of creating the objects that will serve as a layer1_node's communication facilities. Here we create a transmitter and a receiver. We delay the creation of the controller and controller_subprogram until the highest level protocol has been defined.

- **controller_subprogram class csma_prog**

  This subclass provides the essence of our CSMA protocol. Most of the event graph shown in Fig. 7 is implemented here. The protocol is written in the context of the lower level interface which services it – in this case, controller_subprogram that is the interface to *layer1*.

- **csma_prog class link_net_interface**

  This subclass provides the 2-3 interface shown in Fig. 7. The interface itself is written in the context of the lower layer protocol for which it serves as an interface. Thus, it is prefixed with csma_prog.

*Layer2* becomes the new context for the higher ISO layers of our example. The code for *layer2* follows:

```
EXTERNAL CLASS layer1;
layer1 CLASS layer2;
BEGIN
    INTEGER ser_no_counter;
    REAL p_delay;
```

```
REAL PROCEDURE prop_delay(n1,n2); REF(layer1_node)n1,n2;
prop_delay:=p_delay;

PROCEDURE read_delay;
BEGIN
  prompt("Give a value for propagation delay : ");
  p_delay:=inreal;
END;

event_msg CLASS net_msg(len); INTEGER len;;

iso_msg CLASS csma_msg(contents); REF(net_msg)contents;
BEGIN
  INTEGER ser_no;
  real origtime;
  ser_no:=get_ser_no;
  origtime:=time;
END of data_msg;

INTEGER PROCEDURE get_ser_no;
get_ser_no:= ser_no_counter:= ser_no_counter+1;


layer1_node CLASS link_node;
VIRTUAL: PROCEDURE create_objects_at_node;
BEGIN

  PROCEDURE create_objects_at_node;
  BEGIN
    xmtr(1):- NEW transmitter
    (merge_text("xmtr 1 at node ",int_text(id_num)),
    THIS link_node,1);    xmtr(1).identify_cntrl(1);    ACTIVATE xmtr(1);
    rcvr(1):- NEW receiver(merge_text("rcvr 1 at node ",int_text(id_num)),
    THIS link_node,1,0);    rcvr(1).identify_cntrl(1);    ACTIVATE rcvr(1);
  END of create_objects_at_node;

create_objects_at_node;
END of link_node;

controller_subprogram CLASS csma_prog;
BEGIN
  BOOLEAN accessing;
  INTEGER rx_seed;
  REF(msg_q)xq,rq;

  PROCEDURE prologue;
  BEGIN
    print(merge_text("Input data for node ",int_text(station.id_num)));
    prompt("bit rate (bps) : "); set_xmtr_inforate(inreal);
    prompt("retransmission seed: ");  rx_seed:=inint;
    xq:- NEW msg_q(station,"transmit_q",NONE);
    rq:- NEW msg_q(station,"receive_q",NONE);
  END;

  PROCEDURE handle_sq;
  IF NOT accessing THEN access_channel;

  PROCEDURE send;
  BEGIN
    REF(csma_msg)msg;
    IF xq.get(msg,true) THEN
    BEGIN
      turn_on_xmtr;
      start(msg);
    END
    ELSE print("ERROR - no msg in sq");
  END of send;


  PROCEDURE access_channel;
  BEGIN
    REF(msg_q) q;
    REF(csma_msg) msg;
    accessing:=TRUE;
    IF num_prim_signals=0 THEN send
    ELSE ACTIVATE NEW event(THIS csma_prog,
    "csma_prog_access_channel",NONE) delay
    uniform(0.0,1.0,rx_seed),
  END of access_channel;

  PROCEDURE terminate_access; accessing:=FALSE;
```

```
      PROCEDURE msg_sent(msg,xmtr_id); REF(csma_msg)msg; INTEGER xmtr_id;
      BEGIN
        turn_off_xmtr;
        INSPECT station QUA link_node DO
        IF xq.first==NONE THEN terminate_access
        ELSE access_channel;
      END of msg_sent;

      PROCEDURE msg_ret(msg,xmtr_id); REF(csma_msg)msg; INTEGER xmtr_id;
      BEGIN
        Outtext("csma_msg "); Outint(msg.ser_no,4);
        Outtext(" aborted in csma_prog: ");
        Outtext(title); Outimage;
      END;

      PROCEDURE msg_rcvd(msg,rid); REF(csma_msg)msg; INTEGER rid;
      BEGIN
        rq.put(msg);
        ACTIVATE NEW event(THIS csma_prog,"net_prog.handle_rq",NONE);
      END;

      IF evtype="prologue" THEN prologue
      ELSE IF evdestination="csma_prog" THEN
      BEGIN  task;
        IF evtype="access_channel" THEN access_channel
        ELSE IF evtype="handle_xq" THEN handle_xq
        ELSE print("Unrecognized event received by csma_prog");
      END;
    END of csma_prog;

    csma_prog CLASS link_net_interface;
    BEGIN

      PROCEDURE send_msg(msg); REF(net_msg) msg;
      BEGIN
        xq.put(NEW csma_msg(msg.len,msg));
        ACTIVATE NEW event(THIS link_net_interface,"csma_prog.handle_xq",NONE);
      END;

      REF(net_msg) PROCEDURE get_msg(q); REF(msg_q)q;
      BEGIN
        REF(csma_msg)msg;
        IF q.get(msg,TRUE) THEN get_msg:-msg.contents
        ELSE print(merge_text("ERROR - msg NOT available in ",q.title));
      END of get_msg;

    END of link_net_interface;

    read_delay;
  END of layer2;
```

The second block is not a context. Rather, it is our *main* block that defines our higher layer objects (that could not be defined in the physical or link layers) and provides the actions to execute the simulation.

- **net_msg class data_msg**

  We extend the definition of net_msg by defining a subclass that adds the attributes ser_no and origtime. This gives our network layer message the same attributes useful for tracing and statistics collection included in the messages of lower layers.

- **controller class c1_opsys**

  This subclass of controller creates the controller_subprogram with all its subclasses, net_prog being the innermost subclass to be defined. The concatenated object, beginning with controller_subprogram and ending with net_prog, contains all the protocols and interfaces that we have defined. The subclass of controller — c1_opsys — acts as an operating system by scheduling the controller_subprogram for execution. Scheduling in this example is trivial since all that is required is a call to *run_sp* to get things started.

- **link_node class net_node**

  Here we give a node a few more objects that could not be defined in the lower layers — a controller (cl_opsys) and a traffic generator.

- **link_net_interface class net_prog**

  We use this subclass of link_net_interface to build our layer 3-7 receiving protocol, which merely prints out incoming messages.

- **process class traffic_generator**

  The traffic_generator models the sending portion of the layer 3-7 protocol. It also uses the 2-3 interface, in this case to send messages.

The main block ends with several actions that prompt for event tracing, create the node objects, start the simulation, prompt for the simulation period, and reschedule the main block at the end of the simualtion period. The code follows:

```
BEGIN
  EXTERNAL CLASS layer2;
  layer2(10,1,1,1)
  BEGIN
    REAL simperiod;
    INTEGER counter;

    net_msg CLASS data_msg;
    BEGIN
      REAL origtime;
      INTEGER ser_no;
      origtime:=Time;
      ser_no:=counter:=counter+1;
    END;

    controller CLASS cl_opsys;
    BEGIN
      subprg(1):-NEW net_prog
      (merge_text("net_prog FOR cntrl 1 at node ",int_text(station.id_num)),
      station,"net_prog",THIS controller);
      ACTIVATE subprg(1);
      run_sp("net_prog");
    END of cl_opsys;

    link_node CLASS net_node;
    BEGIN
      REF(traffic_generator)gen;
      cntrl(1):- NEW cl_opsys(THIS link_node,1);   ACTIVATE cntrl(1);
      gen:- NEW traffic_generator(cntrl(1).subprg(1));
      ACTIVATE gen;
    END of net_node;

    link_net_interface CLASS net_prog;
    BEGIN

      PROCEDURE prologue;;

      PROCEDURE handle_rq;
      BEGIN
        REF(data_msg)msg;
        msg:-get_msg(rq);
        Outtext("Process msg : "); Outint(msg.ser_no,4);
        Outtext(" at "); Outtext(title); Outimage;
      END;

      IF evtype="prologue" THEN prologue
      ELSE IF evdestination="net_prog" THEN
      BEGIN
        task;
        IF evtype="handle_rq" THEN handle_rq
        ELSE print(merge_text("Unrecognized event in net_prog: ",evtype));
      END;
    END;
```

```
    Process CLASS traffic_generator(interface);
    REF(link_net_interface)interface;
    BEGIN
      REF(data_msg) msg;
      INTEGER tg_seed;
      REAL message_rate;
      REAL message_length;
      prompt("message generator seed : "); tg_seed:=inint;
      prompt("message_rate (#/min.) : "); message_rate:= inreal/60.0;
      prompt("message_length (bits) : "); message_length:=inreal;
      PASSIVATE;
      WHILE TRUE DO
      BEGIN
        HOLD(Negexp(message_rate,tg_seed));
        msg:- NEW data_msg(message_length);
        interface.send_msg(msg);
      END;
    END of traffic_generator;

    PROCEDURE start_simulation; !User should call this at appropriate time;
    BEGIN
      INTEGER i;
      FOR i:=1 step 1 UNTIL numofnodes DO ACTIVATE nodes(i) QUA net_node.gen;
    END of start_simulation;

    PROCEDURE create_objects;
    BEGIN INTEGER i;
      create_phys_chans(1,FALSE);
      chans(1).conn.fill_connectivity_matrix;
      FOR i:=1 step 1 UNTIL numofnodes DO nodes(i):- NEW net_node(i,1,1,1);
    END of create_objects;


    prompt("Number of nodes : "); numofnodes:=inint;
    prompt("Event tracing? (y/n): ");
    IF get_line="y" THEN
    BEGIN
      event_trace:=TRUE;
      prompt("Enter a trace spec: ");
      trace_spec:-get_line;
    END;
    create_objects;
    start_simulation;
    prompt("Enter the simulation period (sec): "); simperiod:=inreal;
    HOLD(simperiod);
  END;
END;
```

END

11-86

DTIC